di Sergio Polini

Problemi di metodo

A gennaio avevamo visto i passi che bisognerebbe compiere per scrivere in modo corretto un programma: nelle due puntate precedenti abbiamo esaminato il funzionamento di QUED in relazione alle tre fasi fondamentali (input, elaborazioni, output), ma così facendo abbiamo solo compiuto il primo passo: «farsi un'idea generale di quello che il programma dovrà fare e delle strutture di dati più adatte». Il mese scorso abbiamo pubblicato un intero «modulo» del programma, quello che si incarica delle operazioni sulle liste circolari doppie, ma ciò non vuol dire che siamo andati oltre quel primo passo, in quanto una struttura di dati non può essere definita prescindendo da come abbiamo bisogno di operare sui dati.
Ora si tratta di andare oltre, ma vedremo che la progettazione «top-down» (sommariamente descritta a gennaio) non è quella sorta di soluzione universale che molti pretenderebbero. Vedremo soprattutto che non esiste un unico metodo «corretto», buono quale che sia il programma che si deve realizzare, ma che è bene disporre di un ampio bagaglio di «metodi» e relative «tecniche»

Avevo intenzione di scrivere QUED seguendo fedelmente la traccia di «edit», il line-editor proposto da Kernighan e Plauger nei *Software Tools in Pascal*. Ma poi ho riscritto tutto da capo: non mi piaceva che in quasi ogni funzione e procedura venisse modificata una variabile «i», rappresentante l'indice del carattere corrente del comando digitato dall'utente. Ho preferito approfittare di alcune caratteristiche del Turbo Pascal (soprattutto le costanti tipizzate) per dividere il programma in «moduli» ben distinti e per nascondere in uno di essi la scansione del comando. Ho usato la versione 3.0 del compilatore per timore di saltare troppo precipitosamente alla versione 4.0, ma, stando a quel che si vede su MC-Link, quasi tutti i turbopascaliani hanno già optato per questa; il programma che vedremo dopo QUED sarà quindi scritto direttamente in 4.0, in modo da poter procedere ancora oltre nell'uso dei moduli grazie alle unit. Preciso comunque che QUED è stato scritto con il 4.0 (per approffitare del TDEBUG 4.0), e che le uniche modifiche che sono risultate necessarie per portare il programma al 3.0 riguardavano la redirezione dell'I/O (vedi puntata di luglio) e il valore della costante MINMEM.

Una volta presa la decisione di riscrivere il programma, mi sono subito imbattuto nei «problemi di metodo».

Top-down e bottom-up

Si racconta la storiella di un celebre matematico che teneva una dotta lezione all'università. Scriveva e scriveva velocemente sulla lavagna formule sempre più astruse, mettendo a dura prova le capacità logiche dei suoi studenti. E non faceva che dire «È ovvio che...». Ma ad un certo punto, dopo il rituale «È ovvio che», si fermò; ripeté il suo intercalare un paio di volte, esitando sempre di più. Poi uscì dall'aula. Tornò una decina di minuti dopo, riprese il gesso, disse «È ovvio che...» e continuò la lezione.

È proprio Plauger che ci ripropone questa storiella, per... dire le cose come stanno: il metodo top-down funziona il più delle volte proprio come quel matematico; in altri termini, per usare le parole di Plauger, il metodo top-down è ottimo per ridisegnare un programma che già si sa come scrivere. Questo non vuol dire che sia «fasullo»: è di grande aiuto per l'esperto, per chi sa come affrontare numerosi diversi tipi di programmi; chi per la prima volta affronta un problema nuovo non può fare molta strada procedendo «dall'alto verso il basso», e deve quindi essere consapevole dell'esistenza di altri metodi.

È facile vedere perché; è sufficiente tornare alla scelta delle strutture di dati: se non so operare su liste concatenate è ben difficile che le scelga per il mio programma. La prima alternativa è il metodo «bottom-up», dal basso verso l'alto, ovvero cercare di individuare fin dal principio le routine di basso livello (quelle che si fanno carico dei «dettagli»), per poi risalire a quelle che le usano, e così via fino al corpo principale del programma. È quello che in fondo abbiamo già fatto: verificare la possibilità di reindirizzare l'I/O, di elaborare i parametri della riga comando, di operare su liste circolari doppie; solo in questo modo abbiamo potuto assicurarci che, ritornati «in alto», avremo potuto effettivamente scegliere certe soluzioni.

In realtà è proprio così che si procede spesso in pratica: top-down per individuare le azioni principali del programma e per poi scomporle fino alle routine di basso livello, e insieme bottom-up per avere le idee chiare su come effettuare quella scomposizione. Si fa top-down quando si scrive sulla lavagna intercalando con «ne segue ovviamente che...», si fa bottom-up quando ci si ritira fuori dall'aula per una decina di minuti.

```
stringa-comando          ::= comando-assoluto | comando-orientato
comando-assoluto         ::= comando-stato | comando-file
comando-stato            ::= char-cmd-stato flag
char-cmd-stato           ::= h | H | P | q | = | S
flag                     ::= n | p | ''
comando-file             ::= char-cmd-file STRINGA
char-cmd-file            ::= e | E | f
comando-orientato        ::= sublista comando-globale | sublista comando-locale
sublista                 ::= indirizzo | indirizzo , indirizzo
indirizzo                ::= riga | riga + spostamento | riga - spostamento
riga                     ::= NUMERO | . | $ | /STRINGA/ | ?STRINGA? | '' | '
spostamento              ::= NUMERO | ''
comando-globale          ::= prefisso comando-ripetibile
prefisso                 ::= g/STRINGA/ | v/STRINGA/
comando-ripetibile       ::= comando-semplice-rip | comando-sostituzione
comando-semplice-rip     ::= char-cmd-semplice-rip flag
char-cmd-semplice-rip    ::= d | n | p
comando-sostituzione     ::= s/STRINGA/STRINGA/ flag | s/STRINGA/STRINGA/ g flag
comando-locale           ::= comando-ripetibile | comando-non-ripetibile
comando-non-ripetibile   ::= comando-semplice | comando-movimento | comando-I/O
comando-semplice         ::= char-cmd-semplice flag
char-cmd-semplice        ::= a | i | c | =
comando-movimento        ::= char-cmd-movimento indirizzo flag
char-cmd-movimento       ::= m | t
comando-I/O              ::= r STRINGA | w STRINGA
```

Figura 1 - La grammatica dei comandi di QUED. I token sono i simboli indicati con un solo carattere o con un nome tutto màiuscolo; fa eccezione la stringa nulla, rappresentata con due apici. Ricordiamo che la barra verticale va letta come un «oppure».

Ma non è tutto. Prima di proseguire, tuttavia, è meglio avere un'idea più precisa del funzionamento di QUED.

I comandi di QUED

I comandi di QUED cominciano con l'indicazione (opzionale) della riga o delle righe su cui operare, mediante numeri di riga o anche attraverso una stringa (ricordiamo che «/pippo/» e «?pippo?» indicano rispettivamente la prima riga successiva e la prima precedente contenente la stringa «pippo»), proseguono con un «prefisso» del tipo «g/<stringa>/» o «v/<stringa>/» (anch'esso opzionale), a cui segue il comando vero e proprio.

Si può indicare una riga anche con la sua posizione relativa ad un'altra; ad esempio «.−3, .+4p» visualizza le righe a partire dalla terza precedente fino alla quarta successiva alla riga corrente (indicata con un punto; ricordiamo che con «$» si può invece indicare l'ultima riga del testo in memoria).

Il prefisso vuol dire: esegui il comando che segue su tutte le righe che contengono (se «g») o non contengono (se «v») una certa stringa. Solo alcuni comandi («d», «n», «p» e «s») possono essere eseguiti globalmente.

Quanto ai comandi veri e propri, possiamo distinguere tra quelli che ammettono e quelli che non ammettono l'indicazione delle righe (dato che usiamo una lista per le righe, chiameremo «sublista» quelle così selezionate).

Ecco i primi:

— «**a**» (append): aggiunge righe dopo una riga indicata (l'immissione termina con una riga rappresentata da un solo punto);

— «**i**» (insert): inserisce righe dopo una riga indicata;

— «**c**» (change): cambia le righe specificate con altre (immesse come con un append);

— «**d**» (delete): cancella le righe indicate;

— «**n**» (number): mostra le righe indicate precedute ognuna dal proprio numero di riga;

— «**p**» (print): mostra le righe indicate;

— «**=**»: mostra il numero della riga indicata;

— «**m**» (move): muove le righe indicate dopo una terza, che va specificata (ad esempio: 3,5m12);

— «**t**» (copy): copia le righe indicate dopo una terza, che va specificata;

— «**r**» (read): legge un file e lo aggiunge dopo la riga indicata;

— «**w**» (write): scrive le righe indicate su un file;

— «**s/<stringa1>/<stringa2>/**» (substitute): sostituisce in tutte le righe indicate <stringa1> con <stringa2>; se seguito da «g» sostituisce tutte le stringhe uguali a <stringa 1>, altrimenti solo la prima di ogni riga.

Alcune note. Se non viene indicata una sublista, vengono assunti degli indirizzi di default: l'ultima riga per «r» e «=», tutto il testo per «w» (come anche per i prefissi «g» e «v»), la riga corrente per gli altri comandi. «r» e «w» possono essere seguiti da un nome di file; se questo manca, si assume o il nome del file indicato nella riga comando (quando si fa partire il programma) o quello specificato con il comando «f», che vedremo subito.

I comandi che non ammettono l'indicazione di una sublista sono:

— «**h**» (help): mostra un messaggio esplicativo dell'ultimo errore;

— «**H**» (Help): come il precedente, ma in più attiva (o disattiva se precedentemente attivata) la produzione di messaggi di errore in chiaro (altrimenti compare solo un punto interrogativo);

— «**P**» (Prompt): attiva o disattiva la visualizzazione di un prompt per i comandi;

— «**q**» (quit): esce dal programma, ma solo se il testo in memoria non è stato modificato o è stato salvato con «w»;

— «**Q**» (Quit): esce comunque dal programma;

— «**S**» (Silent): attiva o disattiva la visualizzazione del numero di righe lette o scritte dopo i comandi «r» e «w»;

— «**e**» (edit): sostituisce il testo in memoria (se non modificato o già salvato con «w») con quello contenuto in un file indicato;

— «**E**» (Edit): sostituisce comunque il testo in memoria;

— «**f**» (file-name): se seguito da un nome di file, fa sì che i comandi che ammettono l'indicazione di un nome di file assumano questo per default, altrimenti semplicemente mostra quale sarebbe il nome di file assunto per default da quei comandi.

Quasi tutti i comandi (fanno eccezione quelli che ammettono l'indicazione di

un nome di file) possono essere seguiti da «n» o «p»: l'effetto è quello di visualizzare dopo l'esecuzione la riga corrente, preceduta o meno dal suo numero di riga.

Destra e sinistra

Proviamo ora a procedere top-down. Possiamo individuare le seguenti azioni: vedere se è indicata una sublista (se sì, se ne memorizzano gli estremi in due variabili globali), vedere se è presente un prefisso «g» o «v» (se sì, si legge la stringa tra le due barre e si marcano le righe che la contengono), leggere il comando con i suoi eventuali argomenti (riga-destinazione, nome di file, le stringhe di «s», ecc.), eseguire il comando globalmente se era preceduto dall'apposito prefisso, altrimenti localmente. Ne avremmo abbastanza per tracciare le linee generali del nostro programma e poi procedere «verso il basso». Il guaio è che si può ben fare così, ma si ottiene un codice in cui la stringa dei comandi digitata dall'utente viene letta e riletta dalla prima all'ultima routine, in cui una variabile globale (la solita «i»), incaricata di registrare la posizione dei caratteri di quella stringa man mano che sono esaminati, viene aggiornata quasi in ogni funzione o procedura. Non è solo «brutto»: in questo modo si rende estremamente arduo il debugging. Perché Kernighan e Plauger abbiano scelto una tale strada rimane per me un mistero; posso solo supporre che non abbiano voluto appesantire un testo già molto denso con l'illustrazione di una vasta gamma di metodi e tecniche. È lo stesso Plauger comunque che, nella sua rubrica «Programming on purpose» su *Computer Language*, riconosce che il metodo top-down dei *Software Tools* non è sufficiente, e illustra diverse alternative.

Oltre alle due direzioni «alto» e «basso», infatti, dobbiamo anche considerare «destra» e «sinistra»: si deve procedere «da destra verso sinistra» quando appare predominante la struttura dell'output, «da sinistra verso destra» quando bisogna in primo luogo venire a capo della struttura dell'input.

Capita di sentir dire che il progetto di un programma deve sempre cominciare dall'analisi dell'output; in realtà così si possono affrontare efficacemente solo alcuni tipi di programmi (ad esempio quelli che portano alla produzione di

```
program Qued;
{$I QDICH.INC}  (* Dichiarazioni di costanti, tipi e variabili globali *)

procedure Q_MsgErrore(Stato: integer);
begin
  writeln(StdErr,'?');
  UltErr := Stato;
  if ErroriInChiaro then writeln(StdErr,Msg[Stato]);
  flush(StdErr)
end;

{$I QLIST.INC}  (* Routine per la manipolazione di liste circolari doppie *)
{$I QFILE.INC}  (* Routine per la lettura/scrittura da/a disco *)
{$I QALEX.INC}  (* Routine per l'analisi lessicale dei comandi *)
{$I QCOMD.INC}  (* Routine per l'esecuzione dei comandi *)
{$I QPARS.INC}  (* Routine per l'analisi sintattica dei comandi *)

procedure Q_Inizializza(var Stato: integer);
var
  i, j, lps: integer;  ps: AnyStr;
begin
  assign(StdIn,'INP:'); reset(StdIn);
  assign(StdOut,'OUT:'); rewrite(StdOut);
  assign(StdErr,'ERR:'); rewrite(StdErr);
  ErroriInChiaro := FALSE; MostraPrompt := FALSE; MostraTotRighe := TRUE;
  Stato          := OK;    UltErr       := 0;     Cambiamenti    := FALSE;
  PromptStr      := '>';   NomeFile     := '';
  i := 1;
  while (i <= ParamCount) and (Stato = OK) do begin
    ps := ParamStr(i); lps := length(ps);
    if ps[1] = '/' then begin
      for j := 2 to lps do
        if ps[j] = 'h' then ErroriInChiaro := TRUE
        else if ps[j] = 's' then MostraTotRighe := FALSE
        else if (ps[j] = 'p') and (j = lps) and (ParamCount > i) then begin
          MostraPrompt := TRUE; i := i + 1; PromptStr := ParamStr(i)
        end
        else Stato := ERRPARAM
    end
    else if NomeFile = '' then NomeFile := ps
    else Stato := ERRPARAM;
    i := i + 1
  end;
  if Stato = OK then begin
    CreaLista(NodoZeroPtr);
    if NodoZeroPtr = nil then Stato := ERRNOMEM
    else begin
      Q_CreaRiga('', NodoZeroPtr, Stato);
      NodoCorrPtr := NodoZeroPtr;
    end;
    if (Stato = OK) and (NomeFile <> '') then Q_ReadFile(NodoCorrPtr, Stato)
  end
end;

begin
  Q_Inizializza(Stato);
  if Stato <> OK then begin
    writeln(StdErr,Msg[Stato]); halt(1)
  end;
  repeat
    CopiaNCPtr := NodoCorrPtr;
    if MostraPrompt then write(StdErr,PromptStr,' ');
    Q_ReadCS(Stato);
    if Stato = OK then Q_ParseCS(Stato);
    if not(Stato in [FINEDATI,OK]) then begin
      Q_MsgErrore(Stato); NodoCorrPtr := CopiaNCPtr
    end
  until Stato = FINEDATI
end.
```

Figura 2 - Il file QUED.PAS

tabulati o alla implementazione di funzioni di «inquiry»): provate a realizzare un programma di sort o un compilatore partendo dall'analisi dell'output! Quanto a QUED, lo stesso inizio del nostro top-down ci ha mostrato che il problema principale è rappresentato dalla interpretazione dei comandi dati dall'utente, ed in questo senso predomina la struttura dell'input.

```
{ QDICH.INC }

const
  FINEAPP = '.';                    { carattere che indica la fine di un append }
  NONTOK  = #0;  FINECMD = #13;  { codici convenzionali ritornati da Q_Token }
  MINMEM  = 256;                 { memoria residua minima in paragrafi di 16 byte }
  { caratteri-comando }
  APPEND = 'a';  NOMFIL = 'f';  MOVETO = 'm';  QUIT   = 'Q';  GLOBNO = 'v';
  CHANGE = 'c';  GLOBSI = 'g';  NUMERA = 'n';  READF  = 'r';  WRITEF = 'w';
  DEL    = 'd';  ERRMSG = 'h';  PRINT  = 'p';  SUBST  = 's';  NUMLIN = '=';
  EDITIF = 'e';  HELP   = 'H';  PROMPT = 'P';  SILENT = 'S';
  EDIT   = 'E';  INS    = 'i';  QUITIF = 'q';  COPYTO = 't';
  { caratteri ausiliari nei comandi }
  RIGACORR = '.';  VIRGOLA  = ',';  ULTRIGA = '$';  ESCAPE = '\';
  CERCANEXT = '/';  CERCAPREV = '?';  PIU    = '+';  MENO   = '-';
  CIFRE     : set of char     = ['0'..'9'];
  { codici di stato e di errore e relativi messaggi }
  OK        =  0;  FINEDATI  = 255;
  ERRNOMEM  =  1;  ERRPARAM  =  2;  ERRIO    =  3;  ERRNUM   =  4;
  ERRCMD    =  5;  ERRSUBLST =  6;  ERRSTRING =  7;  ERRCERCA =  8;
  ERRSINT   =  9;  ERRGLOB   = 10;  ERRNADDR = 11;  ERRSUFF  = 12;
  ERRSPAZIO = 13;  ERRNOMEF  = 14;  ERRNOSALV = 15;  ERRDEST  = 16;
  ERRLUNGI  = 17;  ERRESCAPE = 18;
  Msg: array[1..18] of string[43] =
    ('Memoria insufficiente','Uso: qued [/h] [/s] [/p prompt] [nome_file]',
     'Errore di I/O','Indirizzo non valido','Comando sconosciuto o incompleto',
     'Estremi sublista non validi','Stringa non definita','Stringa non trovata',
     'Errore di sintassi','Comando non eseguibile globalmente',
     'Non ammessa indicazione di sublista','Suffisso non valido',
     'Manca spazio dopo il comando','Nome file non definito',
     'Testo non salvato','Destinazione non indicata o non valida',
     'Stringa risultante oltre 255 caratteri',
     'Un comando non puo'' terminare con ESCAPE');

type
  AnyStr = string[255];
  NPtr   = ^Nodo;
  Nodo   = record
    Prev, Next: NPtr;       { puntatori ai nodi precedente e successivo }
    Txt       : ^AnyStr;    { puntatore ad una riga del testo }
    Glob      : boolean     { vera se oggetto di un comando da eseguire }
  end;                      {   globalmente }

var
  StdIn, StdOut : text;     { standard input e standard output }
  StdErr        : text;     { standard error }
  NodoCorrPtr   : NPtr;     { ptr alla riga corrente }
  CopiaNCPtr    : NPtr;     { copia di NodoCorrPtr }
  NodoZeroPtr   : NPtr;     { fine del buffer }
  Numero        : integer;  { variabili assegnate da Q_Token (in Q_ALEX.INC) }
  Stringa1      : AnyStr;   {      "          "      "     "    }
  Stringa2      : AnyStr;   {      "          "      "     "    }
  NumAddr       : integer;  { numero di righe specificate in un comando }
  Stato         : integer;  { codice di stato }
  UltErr        : integer;  { ultimo codice di errore }
  Cambiamenti   : boolean;  { vera se si e' modificato il testo in memoria }
  ErroriInChiaro: boolean;  { vera se si vogliono messaggi d'errore espliciti }
  MostraPrompt  : boolean;  { vera se si vuole un prompt in "modo comando" }
  MostraTotRighe: boolean;  { vera se si vuole mostrato il num. di righe }
                            {   scritte o lette da un file }
  PromptStr     : AnyStr;   { eventuale prompt; per default: '*' }
  NomeFile      : AnyStr;   { eventuale nome di file associato al testo }
```

Figura 3 - Il file QDICH.INC.

Un po' di grammatica

Procedere «da sinistra verso destra» vuole spesso dire, in pratica, muovere da una descrizione sintattica dell'input, cioè dalla sua grammatica. Una grammatica consente di generare tutte le stringhe appartenenti ad un certo linguaggio attraverso le sue quattro componenti:

— un insieme di *token*, detti anche simboli terminali (quelli di cui sono fatte le stringhe del linguaggio);
— un insieme di simboli non-terminali;
— un simbolo non-terminale assunto come simbolo iniziale;
— un insieme di produzioni; ognuna di queste consta di un non-terminale, di un segno «::=» e di uno o più altri simboli (terminali e non), indicando in questo modo con cosa può essere sostituito ogni non-terminale fino ad arrivare, dopo un certo numero di sostituzioni, ad una stringa di soli terminali.

Non è roba da tutti i giorni, ma non bisogna nemmeno andare troppo lontano: un'appendice del manuale del Turbo Pascal 3.0 riporta la grammatica del linguaggio. Si parte dalla produzione «program ::= program-heading block», si usano le produzioni per «program-heading» e per «block», e si arriva così ad ogni possibile programma Pascal sintatticamente corretto! Possiamo ad esempio sostituire «program-heading» con «**program** program-identifier file-identifier-list» (**program** è in neretto perché è un token), poi «program-identifier» con «identifier», questo con «letter {letter ¦ digit}» (cioè: una lettera seguita da zero o più lettere o cifre); se teniamo presente che «file-identifier-list» può anche essere «empty», cioè una stringa nulla, siamo arrivati alla intestazione del programma: **program** seguito dal nome. Analogamente per «block», che va sostituito con «declaration-part statement-part», e così via.

L'esempio più a portata di mano non è certo il più semplice: vi propongo quindi, accanto ai file QUED.PAS e QDICH.INC, anche la grammatica dei comandi dell'editor. Potrete facilmente verificare che non si tratta di altro che di una traduzione fedele della descrizione informale data prima. A partire dal mese prossimo vedremo cosa fare di quelle 25 righe, vedremo come, procedendo un po' dall'altro e un po' dal basso, e da sinistra verso destra, diventerà facile non solo scrivere il programma, ma soprattutto documentarlo e testarlo.

SILVER SYSTEM

VIA TEVERE, 10 - 16156 GENOVA - TEL. (010) 689324 - FAX (010) 680685

A GENOVA...

HARDWARE & DISTRIBUZIONE

PERSONAL COMPUTER INTERCOMP

XPC30

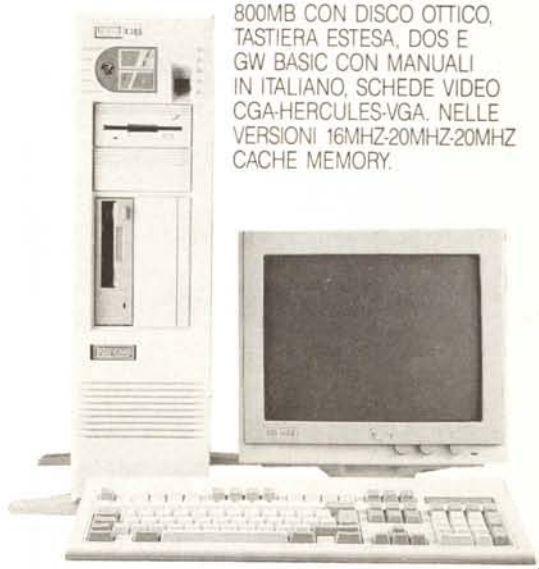
PERSONAL COMPUTER MS/DOS COMPATIBILE CON CLOCK A4.77/10MHZ, ESP. A 640K A BORDO, SCHEDA VIDEO COMPATIBILE CGA-HERCULES SERIALE, PARALLELA, OROLOGIO, MOUSE ADAPTER, SPAZIO FINO A 3 UNITÀ INTERNE DA 3.5" SLIM LINE, (DRIVE 720/1.44MB, HARD DISK 20/40MB) DIMENSIONI CONTENUTE, DOS E GW BASIC CON MANUALI IN ITALIANO. ORA ANCHE NELLA VERSIONE VGA.

XAT

PERSONAL COMPUTER 80286 A 10/12MHZ 1-0 W.S. 512K RAM ESPANDIBILI A 1024 ON BOARD, NUOVO DESIGN A DIMENSIONI CONTENUTE CHE LASCIA SPAZIO A N. 2 ALLOGGIAMENTI DA 5.25" E 2 ALLOGGIAMENTI DA 3.5" SLIM, TASTIERA ESTESA, DOS E GW BASIC CON MANUALI IN ITALIANO. ADATTATORE VIDEO A SCELTA CGA-HERCULES-VGA.

X386

PROCESSORE 80386 1MB ESP. A 2 MB ON BOARD, 2 SLOT ESPANSIONE A 8 BIT, 5 A 16 BIT, 1 A 32 BIT, 7 CANALI DMA, 8/16 PORTE SERIALI (OPZIONALI) 2 ALLOGGIAMENTI DA 5.25" E DUE DA 3.5", HARD DISK DA 20 A 380MB, E FINO A 800MB CON DISCO OTTICO, TASTIERA ESTESA, DOS E GW BASIC CON MANUALI IN ITALIANO, SCHEDE VIDEO CGA-HERCULES-VGA. NELLE VERSIONI 16MHZ-20MHZ-20MHZ CACHE MEMORY.



ANCHE NELLE VERSIONI TRASPORTABILI

ELABORATORI **INTERCOMP** XT-AT-386 NELLE VERSIONI DA TAVOLO, A TORRE, E TRASPORTABILI. SCHEDE VIDEO, CONTROLLER, DI ESPANSIONE, DI COMUNICAZIONE: **SCHEDE MODEM, SCHEDE FAX, SCHEDE DI RETE** SCHEDE DI SISTEMA CPU: **SCHEDE MADRI XT, AT, 386.** PROGRAMMATORI DI EPROM, COPROCESSORI MATEM, LETTORI DI BAR CODE. HARD DISK, STREAMER E GRUPPI DI CONTINUITÀ.

STAMPANTI A MATRICE DI PUNTI E LASERS. **NEC:** TESTINA DI STAMPA 24 AGHI, 360*360 DPI, NELLE VERSIONI B/N E COLORE. DISPONIBILITÀ DRIVERS SOFTWARE PER I MAGGIORI PACCHETTI. **MANNESMANN:** 9 E 24 AGHI, AVANZATA GESTIONE DELLA CARTA, AFFIDABILITÀ NEL LAVORO PESANTE. **PANASONIC:** 9 E 24 AGHI, OTTIMO RAPPORTO QUALITÀ/PREZZO. NUOVA LASER KX-P4450 11 PAGINE/MINUTO. 2 CARICATORI CARTA DI SERIE.

MONITORS MONOCROMATICI E COLORE PER TUTTE LE ESIGENZE E PER TUTTE LE MODALITÀ GRAFICHE E DI TESTO. DISPONIBILITÀ MONITORS MULTISYNC (COMPATIBILITÀ CON TUTTI GLI STANDARDS ATTUALI E FUTURI) ANCHE MONOCROMATICI. MONITORS PER PS/2, AMIGA, MACINTOSH. MONITORS DEDICATI ALLE SCHEDE VGA SIA B/N (PREZZO MOLTO ACCESSIBILE) CHE A COLORI. MONITORS PER CAD E DTP.

?

NEWS
HARD DISK 40MB 3.5" 19 MILLISECONDI. HARD DISK REMOVIBILE DA 20MB. SCHEDA VGA EMULAZ. CGA, HERCULES, EGA, SUPEREGA, VGA. HANDY SCANNER A 1/2 PAGINA CON POTENTE SOFT DI GESTIONE, COMPATIBILE CON VENTURA, PAGEMARKER, PAINTBRUSH, ECC.. SCHEDA DIGITALIZZAZIONE AUDIO AD UTILIZZO PROFESSIONALE, UTILIZZABILE IN OFFICE AUTOMATION CON SOFTWARE VERTICALIZZATO.